An Implementation of a Distributed Interactive Graphics System for a Supercomputer Environment.

RND-87-001

Diana Choi and Creon Levit
NASA Ames Research Center

Abstract

We describe the design, implementation, and performance of a distributed interactive graphics software system. The software is distributed between two machines with very different capabilities, a supercomputer (Cray-2) and a graphics workstation (Silicon Graphics Iris). Both machines run the same operating system (unix) and are programmed by the user. Three dimensional graphical transformations, display updates, the user interface, and related operations are handled by the graphics workstation, while computationally intensive operations and some graphics are performed by the supercomputer. Communication between the two machines consists of a remote graphics protocol using TCP/IP as the transport layer.

Keywords: distributed graphics, computer graphics, workstation, supercomputer, unix, computational fluid dynamics, flow visualization.

Introduction

The traditional method of using computer graphics at a supercomputer center can be an awkward and time-consuming process. To display data generated from codes running on the supercomputer, the user must usually perform the following tasks:

1. Execute the application program that generates data.
2. Convert data from the supercomputer's format to display device format.
3. Download the data to the graphics display device.
4. Execute a program to display the graphics on that device.
5. If the results are unsatisfactory or another iteration is required, modify some parameters and start again.

Users will often do a calculation locally on the workstation, where it takes several minutes or more, rather than on the supercomputer, where it would take only seconds, simply to avoid the overhead of running communication, file transfer, and data conversion programs.

Calculations that require large amounts of memory or are computationally intensive should only be done on a supercomputer. Our application, three dimensional turbulent flow simulation, is an example. Similarly, there are graphics-related calculations

that should only be done on a graphics workstation, utilizing its special purpose graphics hardware. Depending on the workstation, possible examples are three-dimensional viewing transformations, line drawing, polygon filling, and user interface management.

However, it is not always clear whether certain calculations should be performed on a supercomputer or on a workstation. Changes in the data transfer rates between machines and, perhaps more importantly, changes in the way program control is transferred between machines, can profoundly affect the optimal distribution of these calculations.

We describe an implementation that allows calculations to be distributed in an efficient manner, taking advantage of the unique hardware capabilities of each machine.

First, we briefly discuss the hardware configuration of our network and the software base on which the remote procedure calls that implement our distributed graphics system is built. Then, the actual implementation of our distributed graphics protocol is described. Performance measurements of graphics programs distributed between the Cray-2 and the Iris workstation are compared with performance measurements of those same programs running solely on the workstation. A real, successful distributed graphics application is described. Finally, we compare some research related to ours and discuss future plans.

Hardware Configuration

NASA's Numerical Aerodynamic Simulation Program Processing System Network [1] (NPSN) is a national facility for computational fluid dynamics research. That portion of the NPSN hardware relevant to our discussion is the supercomputer, the graphics workstations, and the network communication devices.

The supercomputer is a Cray-2 [2], with 268 million 64-bit words of directly addressable main memory. It has four CPUs with a 4.1 nanosecond global clock. Large matrix multiplications have been benchmarked on the machine at well over 1500 million floating point operations per second. The machine also has a fairly large amount (48,000 megabytes) of high performance disk storage. All communication between the Cray-2 and users is through front end machines, such as workstations, connected by a local area network.

The workstations are Silicon Graphics Iris 2500's and 2500T's [3]. Each Iris has a floating point accelerator, four megabytes of memory, a 400 megabyte disk, a mouse, 24 bit planes, a frame buffer, and network hardware interfaces [1] . Most importantly, each has a pipeline of six special purpose VLSI chips called "geometry engines" that perform graphics calculation. Three dimensional coordinate transformations, perspective projections, and six plane clipping are all performed by the geometry engine. The geometry engine pipeline can transform, clip, and project approximately 60,000 three-dimensional coordinates per second.

The Cray-2 and the workstations communicate over our local area network, a Net-

[1]The 2500T differs from the 2500 in two significant ways: it has a 16 Mhz 32 bit 68020 CPU rather than an 8 MHz 16 bit 68010, and it has a faster floating point accelerator.

work Systems Corporation hyperchannel [4]. The hyperchannel is a four trunk prioritized CSMA/CA bus system with a 50 MHz carrier. Each computer is attached to the hyperchannel trunks via a hyperchannel adapter. During a large data transfer, the link level microcode in the participating adapters reserves both adapters for the duration of the transfer. The Cray-2, which must support many users, has three adapters of its own, while the workstations share adapters, with four workstations interfaced to a single adapter.

Standard System Software

All of the machines on the NPSN run Unix. They all roughly conform to AT&T Unix System V [5], with the addition of Berkeley Unix sockets [6] and the TCP/IP network protocol [7]. The standard commands for file transfer and remote login between machines use TCP/IP, as does our distributed graphics system.

The Standard Iris Graphics Library

Non-distributed user programs running locally on the workstation use the Iris Graphics Library (IGL), supplied by Silicon Graphics Inc. The IGL implements standard graphics primitives, such as move, draw, polygon fill, and viewing transformations. IGL routines also change the color maps, read the mouse, and manipulate windows.

IGL commands may be executed either in immediate mode, where their effects occur right away, or they may be stored in the Iris's memory grouped as a display list object. Each display list object is comprised of a list of commands, with each command consisting of:

$$\langle \textit{subroutine address} \rangle \langle arg_1 \rangle \langle arg_2 \rangle \ldots$$

Display list objects may be interpreted later by invoking the IGL routine `callobj`. Objects may have embedded `callobj` commands. Thus, hierarchical display list objects are supported.

The Original Remote Graphics Library

The first incarnation of the Silicon Graphics special purpose graphics hardware was as a terminal, not as a standalone workstation running Unix. This Iris terminal runs only one program - a standalone interpreter for graphics commands. These graphics commands are received over a network interface from a mainframe, typically a VAX or an IBM. All user graphics programs are written on the mainframe and are linked to the "Remote Graphics Library" (RGL), supplied by Silicon Graphics for that particular mainframe.

The terminal user makes calls in his mainframe program to "graphics" routines like move and draw. The RGL, linked to the user's program translates these calls into tokens and sends these tokens, along with the parameters to the call (e.g. coordinates), over a communication link to the Iris terminal. The terminal then interprets these commands.

Display list objects can be built on the terminal using the RGL routine `makeobj`, and may later be displayed by using the RGL routine `callobj`. Certain RGL calls cause data, such as the mouse position, to be returned from the Iris terminal.

When Silicon Graphics released the Iris workstation, one issue was backwards compatability with the Iris terminal. For this reason, they supply a program for the workstation called **wsiris**, whose function is to emulate an Iris terminal. It listens for tokens coming in over the network, and then does essentially what the Iris terminal would do in response to these tokens. The same RGL is still used by remote mainframe programs. Of course, the Iris workstation can run also user-written graphics programs without a mainframe at all.

RGL On The Cray-2

Our first idea for distributed graphics was to implement the Remote Graphics Library on the Cray-2, and give the user the capability of making calls to RGL from FORTRAN, as well as C. This involved changing the RGL and **wsiris** code to use TCP/IP and Berkeley sockets rather then the XNS protocol used by Silicon Graphics, and adding new routines to convert Cray numeric data to the Iris's internal form.

After this was done, we could run programs already written for the Iris workstation on the Cray-2 (in conjunction with running **wsiris** on the workstation). However, there are problems with this approach.

Most programs written for the Iris workstation are heavily interactive. For example, one might read the mouse continuously at times to implement real time rotation or dragging. When using the RGL-**wsiris** combination, the response time for rotating a simple figure using the mouse is much slower than if the program were running solely on the Iris. To remotely update the viewing transformation based on the mouse position, the exchange in Table 1 has to occur. The problem is that this process is very slow, and the Cray-2 is doing no useful work. Exchanging several small packets between the Cray and Iris to do a simple task like rotation is inefficient.

An optimal situation is one in which heavy computation is done on the Cray-2, a limited amount of distributed graphics is done using RGL, while both user interactions and graphics not requiring intense calculations are handled by the workstation.

If the Cray-2 must do a lot of graphics, it should "batch" large numbers of RGL calls (that don't require return values), so that they are sent to the Iris in a continuous stream of maximum size packets. Although our measured data transfer rates are relatively high (200-500 KBytes per second), we still find that the overhead introduced by system calls for I/O is significant. Exchanging small packets back and forth between machines can greatly reduce response time at the workstation. Hence, the RGL buffers as many graphics commands as it can before making a system call to send the data over the network.

Enhancements To The RGL

Cray-2	Iris workstation
	`wsiris` sleeps waiting for RGL token
call RGL-get-mouse-position	
send RGL token	
send tcp packet	
sleep waiting for mouse-position	
	*acknowledge tcp packet
	*wakeup `wsiris` program
	interpret RGL token
	call IGL-get-mouse-position
	send mouse position to cray
	*send tcp packet
	`wsiris` sleeps waiting for RGL token
acknowledge tcp packet	
wakeup RGL user program	
RGL-get-mouse-position returns	
call RGL-rotate	
send RGL token	
send tcp packet	
	*acknowledge tcp packet
	*wakeup `wsiris` program
	interpret RGL token
	call IGL-rotate

Table 1: Updating the viewing transformation of an object based on mouse position (the slow way). Time flows towards the bottom of the page. Actions marked with a "*" take place in the unix kernel.

If possible, all real time graphics manipulations not requiring intensive calculations, e.g. menu selection or rotation of objects stored in the display list, should be done solely on the Iris workstation. What we desired was a system where control could pass smoothly from the Cray-2, which would use the RGL to build objects on the Iris, to the Iris, where they could be manipulated locally.

A key idea was to turn the RGL token interpreter program `wsiris` into a subroutine, called `rcvgraphics`, that any Iris program can call. The `rcvgraphics` subroutine continuously interprets graphics tokens received over the network, by calling the appropriate

IGL routine. It returns when it receives a special token [2] from the Cray-2; a routine, named `last_one` was added to the Cray-2 RGL to send this special token. Thus a model distribued application can be implemented as shown in Table 2.

Cray-2	Iris workstation
compute	initialize colormaps, draw menu ...
	`rcvgraphics()`
`makeobj(1)`	
`move()`	
`draw()`	(`rcvgraphics` is interpreting tokens and
`draw()`	calling the IGL to build a display
`move()`	list object.)
`draw()`	
...	
`endobj(1)`	
`last_one()`	
	`rcvgraphics` returns.
compute	`callobj(1)`
	`read_mouse()`
	`rotate()`
	`callobj(1)`
	...

Table 2: A model application, where the Cray-2 and the Iris both do graphics.

Such a cycle cycle can be repeated, with the Cray-2 making another object, and the Iris transforming it in real time in response to user input. The Cray-2 is freed from handling events like mouse interrupts, and the Iris is freed from computing the object's initial shape.

Synchronous message passing routines were added to both the remote and local graphics libraries to provide additional flexibility in controlling dustributed applications. These routines (`sndmsg` and `rcvmsg`) allow exchange of unformatted ascii data, which is interpreted by the given application. The most common application for `sndmsg` and `rcvmsg` is one in which the Iris tells the Cray-2 which computational task to perform after querying the user. An example of this sequence is shown in Table 3:

[2] Using this technique, the `wsiris` program, mentioned above, would be written (in C) on the Iris simply as:

```
main()
{
        rcvgraphics();
}
```

In this case, as with the original `wsiris` program, any mouse or keyboard input to the program must be handled by the Cray-2.

Cray-2	Iris workstation

`rcvmsg(message)`
`if (message = ''1'')`
`  call subroutine1`
`elseif (message = ''2'')`
`  call subroutine2`
...
	get user command
	`sndmsg(command)`

Table 3: A typical use of `sndmsg` and `rcvmsg`.

Building Display Lists On The Cray-2

In an effort to speed up distributed graphics, we investigated the feasibility of building the display list objects on the Cray-2 and sending complete objects to the Iris, rather then having `rcvgraphics` call the IGL to build objects on the Iris as it interprets RGL tokens.

Because Iris display list commands are 32-bit addresses, whereas RGL tokens are 16 bits long, a display list object consisting only of three dimensional moves and draws (as most of our are) is 14% larger than the corresponding RGL data describing the same object. [3]

The justification for building objects on the Cray-2 is that it is faster than building them on the Iris in response to RGL tokens, and that this more than makes up for the overhead of transmitting 14% extra data. This proved to be true for the 2500s, but not for the faster 2500T's, as is shown by the benchmark results below.

Benchmark Results

Performance measurements were made using two programs. Each program was distributed between the Iris and the Cray-2 and, for comparison, was also run totally locally to the Iris. In all cases, the Iris was dedicated to our programs, (no other users), but the Cray-2 was in multi-user (production) mode. To eliminate variations caused by the Cray-2 system load, the minimum times are reported.

The first program requires almost no calculations and consists primarily of a series of two-dimensional moves and draws. It was run in both immediate mode, where the moves and draws produce immediate graphical results, and in display list mode, where they are encapsulated in an object that is later interpreted and displayed. In the distributed versions of this program, display list objects were built using 2 methods: on the Iris using RGL, and on the Cray-2 where they are then sent to the Iris. The results are shown in Table 4.

Since the only difference between the 2500 and the 2500T is the CPU (the same special-purpose graphics hardware is on both machines), it is reasonable to assume that

[3] The data (parameters) associated with commands and tokens is identical.

	2500 (MC68010)	2500T (MC68020)
Local to Iris - immediate mode:	1.98	1.96
Local to Iris - display list mode:	3.85	2.55
Distributed - immediate mode:	4.32	1.98
Distributed - display list objects built with RGL:	7.61	3.50
Distributed - display list objects built on Cray-2:	4.52	3.50

Table 4: Running times, in seconds, of a simple program that does a lot of graphics but almost no numerical computation.

only a slight improvement will be seen in tasks that are not CPU-bound. This is verified by the figures above on local immediate mode, where both the 68020 and the 68010 are fast enough to keep the graphics pipline full.

In both local display list mode and distributed RGL display list mode, the Iris must spend time creating display lists. Here, we see that a faster CPU does make a difference.

Distributing tasks between machines introduces the additional overhead of data transfer and system calls for I/O on each machine. Use of display lists, either local or distributed, also introduces overhead since they must be built on some machine and then later interpreted on the Iris.

However, in our distributed graphics system, operations can proceed concurrently on the Cray-2 and the Iris. We can overlap the generation of graphical data on the Cray-2 with its interpretation and display on the Iris. Overlapping large amounts of graphics in this way also lets us take advantage of our relatively high data transfer rates (200 - 500 Kbytes/second) and large network I/O buffers (16K bytes).

A Successful Application

The second program we report on is a fairly complicated application for interactive viewing of three dimensional vector fields. It is the Real-Time Interactive Particle Tracer (RIP) [9].

RIP takes as input a vector field and a set of starting locations in the field, along with a computational grid [4] . For each specified starting location in the field it computes and displays the "particle trace" (integral curve) corresponding to that point [10] . It uses an explicit Euler method to advance the particle in space, based on interpolated values of the vector field inside a computational grid cell.

This program is computationally intensive, consisting of a section of floating point calculations followed by a series of moves and draws. In the distributed case, RGL tokens are used to construct the objects on the Iris. Results are shown in Table 5.

The dramatic decrease in response time gained by distributing this three dimensional

[4] A (set of) three dimensional curvilinear coordinate system(s) and boundaries, see [8].

	2500 (MC68010)	2500T (MC68020)
Local to Iris - display list mode:	275.0	70.0
Distributed - display list objects built with RGL:	12.0	4.0

Table 5: Running times, in seconds, of a program that does both significant numerical calculation and significant graphics. This program takes 3.1 seconds to run on the Cray-2, neglecting distributed graphics.

flow visualization program between the two machines has been accepted with enthusiasm by our users, and has led them to take this distributed graphics library quite seriously. However, faster response is not the only benefit obtained. The vector fields that this program manipulates can be very large (over 160 million bytes) and must be kept in memory. This is essentially impossible for a program running locally to the Iris, but is not much of a problem for the distributed version, since all operations on the vector fields take place solely on the Cray, with only the selected particle paths being sent to the Iris.

Three dimensional boundary surfaces are displayed on the Iris for reference along with the particle traces. Three dimensional rotation, translation, clipping, menu interaction, and three dimensional particle starting point selection is all mouse driven and handled locally on the Iris.

A videotape of this application being used is available from the authors.

Related Research

Research into distributed graphics has a significant history. Hartzman [11] does a relatively complete survey of the field. Our work is similar and [12] in that we use the same language and operating system on both processors, and in that the user need not start with a partitioned application. (Indeed, most of our significant applications to date started out as local Iris only programs, and were later distributed.) However, unlike [12], our user does need to specify *some* of the cross partition interface. Specifically, our user must encode and decode the non-graphical part. The CAD/CAM community has been using distributed graphics for some time [13] and protocols for distributed ray tracing are now in use [14].

The Future

Unlike so much previous work in distributed graphics, we have finally gotten around the data transfer bottleneck. Our system is well balanced, and we can tune applications so that compute time, data transfer time, and graphics processing time contribute about equally.

In the future, we will experiment with a more general remote procedure call mechanism between the Cray-2 and the Iris [15], along with instrumentation [16] and a stub generator for semiautomatic distribution of user applications [17] [18]. We also look

forward to seeing additional computationally intensive user applications (such as three dimensional grid generation [8] and contouring) distributed using the current libraries.

Conclusions

We have described the implementation of a distributed graphics system, running between a workstation and a supercomputer. Our implementation is successful and in constant use. Because the same system call interface (unix), communications protocols (TCP/IP), and languages (C and fortran) were available on both machines, the implementation was relatively straightforward. Distributed graphics programs can run on our local network, or over the internet to distant sites. Applications can be split so that intensive computation and graphics are done by the supercomputer while user interaction, viewing transformations, and additional graphics are done by the workstation.

Acknowledgements

We would like to thank Gordon Bancroft, Pieter Buning, Tom Lasinski, Fergus Merritt, and Stuart Rogers for helping to make this paper possible.

References

[1] F. R. Bailey. "Status and Projections of the NAS Program." to appear in *Symp. on Future Directions of Computational Mechanics (ASME winter Annual Meeting)*. Anaheim, CA, Dec. 7-12, 1986.

[2] Cray Research Inc. *Cray-2 Engineering Maintenance Manual*. Cray Research Inc., Mendota Heights, MN, 1985.

[3] Silicon Graphics Inc. *IRIS User's Guide*. Silicon Graphics Inc., Mountain View, CA, 1986.

[4] Network Systems Corp. *Nucleus Adapter Reference Manual*. Network Systems Corp., Brooklyn Park, MN, 1981.

[5] D. E. Kevorkian, ed. *System V Interface Definition*. AT&T, Indianapolis, IN, 1285.

[6] S. J. Leffler. *A 4.2BSD Interprocess Communication Primer*. Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, CA, 1983.

[7] E.J. Feinler et al., eds. *DDN Protocol Handbook*. Vol. 1, Defense Technical Information Center, Alexandria, VA, 1985.

[8] J. F. Thompson et al. *Numerical Grid Generation, Foundations and Applications*. Elsevier Pub. Co. Inc., New York, NY, 1985.

[9] S. Rogers, F. Merritt, D. Choi. RIP (Real-Time Interactive Particle Tracer). Computer Program. NASA Ames Research Center, Moffett Field, CA, 1986.

[10] P. G. Buning and J. L. Steger. "Graphics and Flow Visualization in Computational Fluid Dynamics." in *AIAA 7th Computational Fluid Dynamics Conference.* Cincinnati, OH, July 15-17, 1985.

[11] P. D. Hartzan *Configurable Software for Satellite Graphics.* research and development report COO-3077-148, Courant Mathematics and Computing Laboratory, New York, NY, 1977.

[12] G. Hamlin. *Configurable Applications for Satellite Graphics.* Ph.D. Dissertation, University of North Carolina, Chapel Hill, NC, 1975.

[13] J. R. Rao et al. "Performance Evaluation of a Test Distributed Graphics System" in *IEEE Computer Software and Applications Conference.* Chicago, IL, 1979.

[14] Mike Muss, U.S. Army Ballistics Research Laboratory. Personal Communication. 1986.

[15] Sun Microsystems. *Remote Procedure Call Protocol Specification.* Sun Microsystems, Mountain View, CA, 1985.

[16] S. L. Graham et al. "Gprof: a Call Graph Execution Profiler." in *Proc. ACM SIGPLAN '82 Symposium on Compiler Construction.* Association for Computing Machinery, New York, NY, 1982.

[17] R. J. Souza and S. P. Miller. *Unix and Remote Procedure Calls: A Peaceful Coexistence?* MIT Project Athena, Cambridge, MA, 1986.

[18] M. B. Jones et al. "Matchmaker: An interface Specification Language for Distributed Processing." in *Proc. 12th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages.* Association for Computing Machinery, New York, NY, 1985.